Bubble Shooter

Yuvraj Tilotia
CMP501 – Network Game Development
2200518@uad.ac.uk

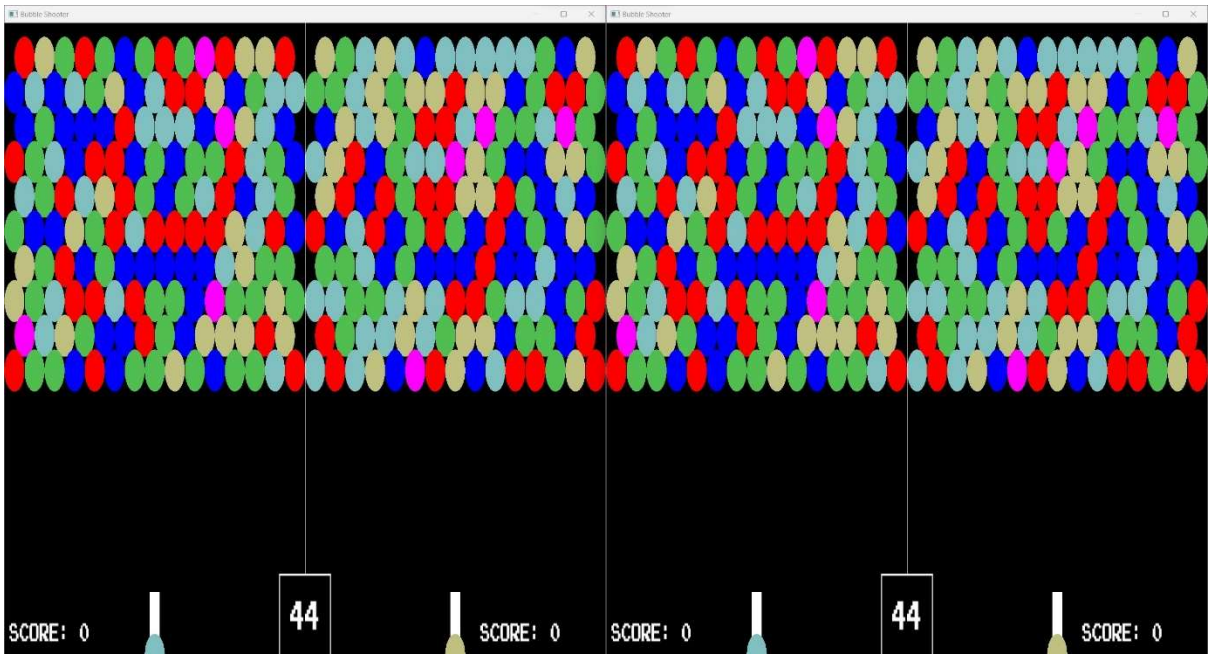

Fig 1 – Bubble Shooter

1. ARCHITECTURE

Given the requirements for this project, the initial goal was to realize a simple multiplayer game meant for two players, i.e., a bubble shooting game using SFML. After some research on which architecture was more suitable, ultimately the choice fell to the client-server architecture. The reason behind this choice was relatively dictated by the amount of prior knowledge and experience that was withheld on the subject, compared to what was known on peer-to-peer architectures. Finally, another important reason that was held into thought was how client-server architectures are commonly considered the best choice for action games, in comparison to peer-to-peer architectures. [1]

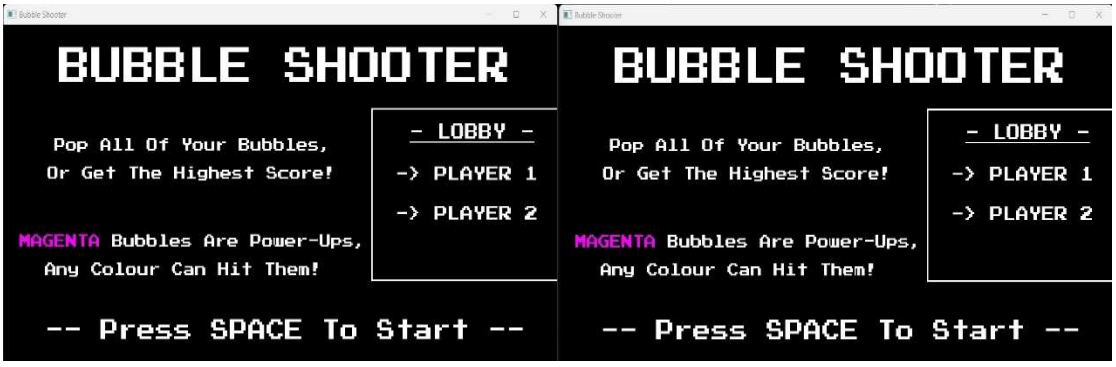

Fig 2 – Main Screen

2. PROTOCOLS

Upon running the game, the first player will be assigned both the server and client role and the second player will only have the client role. There is a dependency of the second player on the first player. If any of the players quit the game, the game also quits for the other player.

We chose TCP (Transport Layer Protocol) for this project. The reasons behind this choice were –

a) Small amount of data transfer ->
   There are small data transfers between the client and the server. This feature of the game favours TCP over UDP.

b) Negligible latency issues ->
   For latency below 100ms, both TCP and UDP pose no issues.

c) No real time error handling ->
   The game has simple mechanics. So, it won't require any high level of error handling. This feature gives TCP the edge over UDP as it is easy to manage errors, packet loss in TCP.

TCP makes sure that every packet sent is delivered, but this does cause a certain amount of overhead, and technically the most important packet that we need for the proper functioning of the game is the last one that has been sent which signals the bubble hitting the set bubble. Any packet loss after this state will not affect the result.

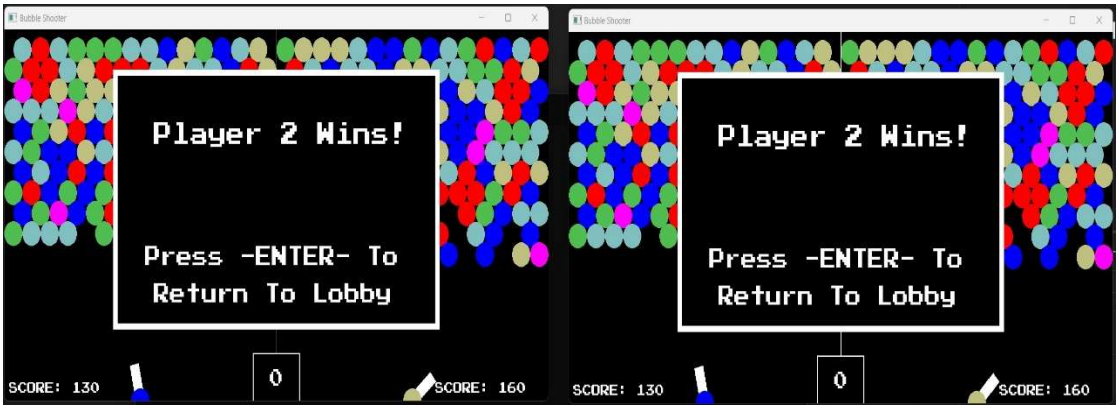

Fig 3: Win Screen

3. APPLICATION PROGRAM INTERFACE (API)

The API chosen to develop this project is the Simple and Fast Multimedia Library. [2] It was considered suitable for the scope of this assignment as it is not only used commonly as a game-building tool for small-scale indie projects but does also feature a networking module that is easy to work with. Here are some of the classes from the module that were used for the game:

a) sf::IpAddress ->
   This class encapsulates an IPv4 network address. Among its features, the most notable one is the support for multiple address representations.

b) sf::Packet ->
This utility class can build blocks of data to transfer over the network and provides a safe and easy way to serialize this data, as well as a wrap and unwrap it, to send it overusing sockets.

c) sf::TcpListener ->
This class represents a listener socket, a special type of socket that is used in TCP connections and listens to a given port and waits for connections on that port.

d) sf::TcpSocket ->
This class is quite trivial; it simply represents a socket running over TCP.

e) sf::RenderWindow ->
This class enables a window that can serve as a target for 2D drawing. sf::VideoMode is used to set up the windows at the time of creation.

f) sf::Event ->
This class holds all information about a system event that just happened such as key press, mouse movement, etc.

4. INTEGRATION

Integrating the networking module with the rest of the application has been quite an easy process since the entire game was built using the same library. These methods are: [3]

a) void TCPServerListener() ->
This method is launched on a separate thread and is used by the server to listen for connections using the sf::TcpListener class from SFML. The only role of the listener class is to wait for incoming connection attempts on a given port, as it can't send or receive data.

b) void TCPClientListener() ->
This method is a variant of the function meant for the client. It differs from the first one as it doesn’t use a Listener class; after all the client doesn’t need to wait for a connection attempt on any port, since it is the one attempting to connect: it is assumed that the client knows already the server’s IP address. This method is like the previous one only concerning how the receiving loop works.

c) void TCPServerSend() ->
This method is used to send the game information from the server to the client. It locks access to the packet to avoid any accidental packet overwriting and sends the information over TCP before unlocking the resources.

d) void TCPClientSend() ->
As before, this is the client version of the previous method and sends information about the client to the server. The other important difference from the server’s sending method

is that the client doesn't need to send any information about the bubble's location, or the bubble's angle, as the server is authoritative about this information.

5. PREDICTION

During draft of the game, we tried to predict the bubble attributes of position and direction by using a bubble angle which would be calculated at random times during the game. This bubble angle would be calculated whenever the bubble is shot. It'll also hold the boundary data, so that we could predict collision. This was then changed from random to periodic.

The other prediction was to make sure collisions between the same colour bubbles is recognized. This was proof checked by using a boundary on each bubble, which is coded to be the same colour as the bubble. So, when two bubbles of same colour collide, the bubbles can be cancelled out.

We used laws of reflection as well to predict the movement of the bubbles upon colliding with the screen boundaries. We calculated the angle at which the bubbles collided and then added 90 degrees to it so that we can have a normal reflection.

Upon packet loss or drops, the shooter would remain in the same state as it was before the drop: if stationary, then remain stationary. If moving left to right, move left to right. If moving right to left, move right to left. Using Clock function, we could exactly predict the direction of shooter for the other player's screen.

6. TESTING

The game's network has been tested by using Clumsy, [4] a useful tool that can simulate lag, drops, out-of-order packets, and other networking problems in an interactive manner.

The first test error created was lag. Below 50ms latency, there is no visible difference. Once the latency is raised to more than 300ms, there is visible difference in the response. This does not mean, that the game underperforms. The prediction models predict the direction of the shooter perfectly, perform collision smoothly.

The second test error made was packet drops. For this case, TCP proved inadequate as it favours to wait for the full packet and then complete the transfer.

The third test error was re-ordering packets. As we were using TCP, it made sure to re-arrange all the packets in correct order and then transfer.

The main issue was dealing with packet's integrity. Whenever the packet's integrity was compromised, the game would crash.

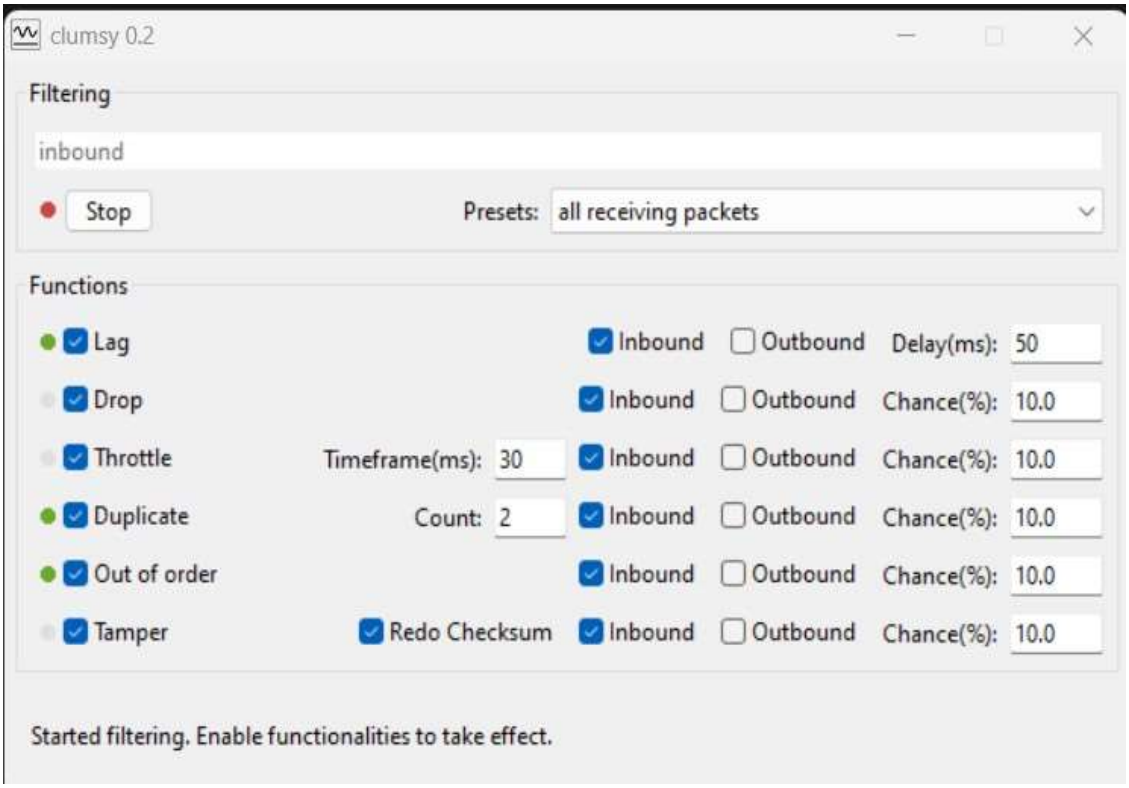

Fig 4: Clumsy Testing Tool

7. REFERENCES

[1] What every programmer needs to know about game networking (2010). Available at:

https://gafferongames.com/post/what_every_programmer_needs_to_know_about_game_networking/ (Accessed: 30 November 2022).

[2] SFML Documentation (no date). Available at:

https://www.sfml-dev.org/documentation/2.5.1/ (Accessed: 15 December 2022).

[3] Winsock (no date). Available at:

https://learn.microsoft.com/en-us/dotnet/api/system.net.sockets?view=net-7.0 (Accessed: 16 December 2022).

[4] Clumsy (no date). Available at:

http://jagt.github.io/clumsy/ (Accessed: 3 January 2023).